PCT WORLD INTELLECTUAL PROPERTY ORGANIZATION
International Bureau

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 3/06 | A1

(11) International Publication Number: WO 99/15953

(43) International Publication Date: 1 April 1999 (01.04.99)

(21) International Application Number: PCT/US98/18441

(22) International Filing Date: 3 September 1998 (03.09.98)

(30) Priority Data:
08/936,302 24 September 1997 (24.09.97) US

(71) Applicant: SONY PICTURES ENTERTAINMENT, INC. [US/US]; 10202 W. Washington Boulevard, Culver City, CA 90232 (US).

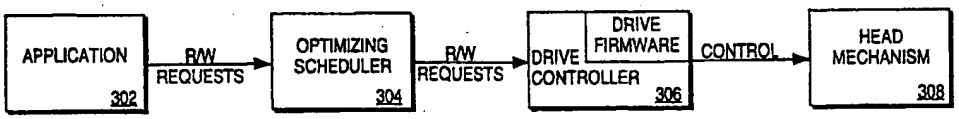
(72) Inventor: OLIVER, Richard, Joseph; 21629 Ocean Vista Drive, Laguna Beach, CA 92677 (US).

(74) Agents: SOBRINO, Maria, E. et al.; Blakely, Sokoloff, Taylor & Zafman, 7th floor, 12400 Wilshire Boulevard, Los Angeles, CA 90025-1026 (US).

(81) Designated States: AL, AM, AT, AT (Utility model), AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, CZ (Utility model), DE, DK, DK (Utility model), EE, EE (Utility model), ES, FI, GB, GE, GH, GM, HR, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SK (Utility model), SL, TJ, TM, TR, TT, UA, UG, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GW, ML, MR, NE, SN, TD, TG).

Published
With international search report.

(54) Title: OPTIMIZING SCHEDULER FOR READ/WRITE OPERATIONS IN A DISK FILE SYSTEM



(57) Abstract

A system for optimizing the order of read/write commands in a disk file system is described. An optimizing scheduler blocks the transmission of disk access requests from an application program to a disk controller until a relatively large set of read/write requests is collected. The scheduler then sorts the set of read/write requests into an order which corresponds to the physical distribution of sectors on the disk accessed by the set of read/write requests.

FOR THE PURPOSES OF INFORMATION ONLY

Codes used to identify States party to the PCT on the front pages of pamphlets publishing international applications under the PCT.

AL	Albania	ES	Spain	LS	Lesotho	SI	Slovenia
AM	Armenia	FI	Finland	LT	Lithuania	SK	Slovakia
AT	Austria	FR	France	LU	Luxembourg	SN	Senegal
AU	Australia	GA	Gabon	LV	Latvia	SZ	Swaziland
AZ	Azerbaijan	GB	United Kingdom	MC	Monaco	TD	Chad
BA	Bosnia and Herzegovina	GE	Georgia	MD	Republic of Moldova	TG	Togo
BB	Barbados	GH	Ghana	MG	Madagascar	TJ	Tajikistan
BE	Belgium	GN	Guinea	MK	The former Yugoslav Republic of Macedonia	TM	Turkmenistan
BF	Burkina Faso	GR	Greece			TR	Turkey
BG	Bulgaria	HU	Hungary	ML	Mali	TT	Trinidad and Tobago
BJ	Benin	IE	Ireland	MN	Mongolia	UA	Ukraine
BR	Brazil	IL	Israel	MR	Mauritania	UG	Uganda
BY	Belarus	IS	Iceland	MW	Malawi	US	United States of America
CA	Canada	IT	Italy	MX	Mexico	UZ	Uzbekistan
CF	Central African Republic	JP	Japan	NE	Niger	VN	Viet Nam
CG	Congo	KE	Kenya	NL	Netherlands	YU	Yugoslavia
CH	Switzerland	KG	Kyrgyzstan	NO	Norway	ZW	Zimbabwe
CI	Côte d'Ivoire	KP	Democratic People's Republic of Korea	NZ	New Zealand		
CM	Cameroon			PL	Poland		
CN	China	KR	Republic of Korea	PT	Portugal		
CU	Cuba	KZ	Kazakstan	RO	Romania		
CZ	Czech Republic	LC	Saint Lucia	RU	Russian Federation		
DE	Germany	LI	Liechtenstein	SD	Sudan		
DK	Denmark	LK	Sri Lanka	SE	Sweden		
EE	Estonia	LR	Liberia	SG	Singapore		

OPTIMIZING SCHEDULER FOR READ/WRITE OPERATIONS IN A DISK FILE SYSTEM

FIELD OF THE INVENTION

The present invention relates generally to the field of computer operating systems, and more particularly to scheduling read/write requests in a disk-based file system.

BACKGROUND OF THE INVENTION

Disk based operating systems divide a disk into physical sectors which represent the fundamental units for data storage on the disk. A sector is a portion of the circumference of a track on the disk, and the read/write mechanism within the disk drive can access specific sectors on the disk directly. When an application issues a command to read a specific byte from a disk file, the file system locates the correct surface, track, and sector, reads the entire sector into a memory buffer, and then locates the requested byte within that buffer. Because sector sizes allocated on a disk may be small in relation to the total amount of data required in a typical data transaction, the read/write head may be forced to seek among several different sectors in response to a read or write command by an application program.

The physical read/write mechanism of a hard drive is generally in one of three states: idle, seek, or input/output (I/O). In order to maximize the I/O bandwidth of the disk drive, the time that the drive spends in the idle and seek states must be minimized. A seek state corresponds to moving the read/write mechanism to the correct sector, and the time required by a seek operation is referred to as the seek time.

One method used in present disk drive systems of minimizing idle times is to ensure that new read/write requests are available as the previous read/write operations complete. This essentially creates a queue of read/write requests which minimizes the time gap (idle time) between individual read/write operations. Likewise, a method used in present disk drive systems of minimizing seek times is to analyze the queued read/write requests and perform them in the most efficient order with regard to the order of the sectors as they are accessed on the disk. A shortcoming of these present systems is that because they attempt to minimize the latency between the time a request is initiated and the time the operation is performed, the number of requests that are queued at any given time is also minimized. The minimal queue length consequently limits the number of requests which can be optimally ordered, thus resulting in a less than optimal ordering in cases where the number of requests exceeds the size of the queue.

Many general purpose file systems are suitable for use with applications for which minimal optimization of read/write requests occurs. Certain applications, however, require sets of data transfer operations to disk to be completed within a maximum time limit. This time requirement translates to a minimum disk input/output bandwidth, and is typically measured in terms of number of bytes transferred to or from the disk per unit time. For such applications, excessive seek times may increase disk access times the beyond the minimum bandwidth requirements. Examples of I/O intensive applications include real-time applications which require extensive disk access. Although a common solution to this problem may be to use faster disk drives, design and cost constraints may prevent the use of adequately fast disk drives.

It is therefore an intended advantage of the present invention to provide a system for optimizing a large number of disk access requests in a disk file system for use with a broad range of disk drive devices.

SUMMARY OF THE INVENTION

The present invention discloses a method for increasing the disk I/O bandwidth in a disk based file system. Disk access requests from an application program to a drive controller are blocked until a relatively large set of read/write requests are collected. An optimizing scheduler sorts the order of the requests so that the order of the set of read/write requests corresponds to the physical distribution of sectors on a disk which are to be accessed by each read/write request. The ordered set of read/write requests are then transmitted to the drive controller.

According to one embodiment of the present invention the disk input/output queue is blocked from being processed for a fixed period of time while read/write requests accumulate in a buffer. The optimizing scheduler then optimizes the entire queue in a single pass and sends the operations to the drive controller in the optimized order.

An embodiment of the present invention effectively increases the size of the queue of read/write requests to the disk drive, so that new requests inserted into the queue have a greater chance of being optimally placed between requests already in the queue, thus improving the average seek time for the set of requests.

Other features of the present invention will be apparent from the accompanying drawings and from the detailed description which follows.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention is illustrated by way of example and not limitation in the figures of the accompanying drawings, in which like references indicate similar elements, and in which:

Figure 1 is a block diagram of a computer system which may be used to implement an embodiment of the present invention.

Figure 2 illustrates an example of sector distribution on disk media.

Figure 3 is a block diagram of a disk file system which uses an optimizing command scheduler according to one embodiment of the present invention.

Figure 4 is a flowchart illustrating the process of re-ordering sets of read/write requests to optimize disk drive performance according to one embodiment of the present invention.

Figure 5 is a table which illustrates the relative seek distances required in different command queuing systems, including a system according to one embodiment of the present invention.

DETAILED DESCRIPTION

A system for collecting and optimizing read/write commands in a disk file system is described. In the following description, for purposes of explanation, numerous specific details are set forth in order to provide a thorough understanding of the present invention. It will be evident, however, to one skilled in the art that the present invention may be practiced without these specific details. In other instances, well-known structures and devices are shown in block diagram form to facilitate explanation.

Hardware Overview

Figure 1 illustrates a block diagram of a computer which may be used to implement an embodiment of the present invention. The computer system 100 includes a processor 102 coupled through a bus 101 to a random access memory (RAM) 104, a read only memory (ROM) 106, and display device 120. Keyboard 121 and cursor control unit 122 are coupled to bus 101 for communicating information and command selections to processor 102. Also coupled to processor 102 through bus 101 is an input/output (I/O) interface 123 which can be used to control

and transfer data to electronic devices connected to computer 100.

Mass storage device 107 is also coupled to processor 102 through bus 101. Mass storage device 107 represents a memory device which stores data accessible from processor 102 through a sector-based file system used by the computer system 100. Mass storage device 107 may be a persistent storage device, such as a floppy disk drive or a fixed disk drive (e.g., magnetic, optical, magneto-optical, or the like), which can directly access locations on the disk for reading and writing data to and from the disk media. Alternatively, mass storage device 107 could be a tape drive which accesses data blocks placed sequentially on streaming tape media. It should be noted that the architecture of Figure 1 is provided only for purposes of illustration, and that a computer used in conjunction with the present invention is not limited to this specific architecture.

Command Optimization

Disk file systems organize data on disks in sectors. Due to file fragmentation or sector interleaving, data relating to a particular file may be spread among several discontiguous sectors. Figure 2 illustrates an example of the distribution of related sectors on a disk file. Disk 200 contains several tracks and four sectors containing data. The sectors are numbered 1, 2, 3, and 4. If an application requests to read to data from these sectors, a file system which employs a simple command FIFO (first-in, first-out buffer) would simply access the sectors in the order the commands are received. Thus, with reference to Figure 2, if the application requested data to be read from sector 1, sector 2, sector 3, and then sector 4, the read/write head of the disk drive would access the sectors in that order. As can be seen in Figure 2, however, this order requires several seek operations between each read operation since each sector is on a different track, and the sectors are not arranged on disk in the order in which they are requested.

Certain known disk file systems use a command queuing system to minimize the number of seeks between sector read operations by

ordering the sequence of the sectors as they are accessed by the application to correspond to the physical order of the sectors on the disk. For example, given the sector distribution illustrated in Figure 3, an optimal order to the read/write requests would be 1, 4, 3, 2. As can be seen in Figure 2, this order eliminates any extra track crossings between sequential sector accesses.

Present systems of efficiently ordering disk read/write requests, such as command queuing systems, however, provide only a limited queue size in which commands are re-ordered. Queue size is limited because these systems also attempt to minimize the read/write cycle latency which is the measure of the amount of time between the time that a read/write request is initiated and the time that it is performed. As the length of the queue is increased, the latency correspondingly increases. Thus, in the example provided above, if the queue can only accommodate two instructions while an access command is being completed, the system would cause the sectors to be read in the order of 1-3-2-4. Given the distribution of the sectors in Figure 2, this re-ordered scheme does not reduce the number of seek operations from the original order of 1-2-3-4. Although Figure 2 illustrates an example in which only four sectors are accessed, it will be appreciated that most applications request disk accesses which involve many sectors. In these cases, the limitations of the simple FIFO and command queuing systems are amplified.

Figure 3 illustrates a system according to one embodiment of the present invention in which the effective size of the read/write command queue is increased so that more read/write commands are inserted in the queue to be placed in the optimum order. In Figure 3, application 302 generates read/write requests for data on a disk controlled by drive controller 306. Drive controller 306 represents both the controller hardware circuit within the disk drive, as well as the driver firmware executed by the disk drive. Drive controller 306 sends control signals which control the movement and read/write functionality of head mechanism 308.

In one embodiment of the present invention, an optimizing scheduler 304 receives the read/write requests from application 302. Optimizing scheduler 304 collects a number of read/write requests, and then orders these requests so that seek operations between the accessed sectors is minimized.

In a method of the present invention, the optimizing scheduler 304 blocks the queue and holds the read/write requests in a buffer for a specified period of time. This period of time defines a scheduler period which can be varied depending on factors such as the rate of requests issued by the application, the I/O bandwidth requirements of the system, and the amount of fragmentation on the disk. When the end of the scheduler period is reached, the optimizing scheduler optimizes the entire queue in a single pass and sends the requests to the disk in the optimized order.

Alternatively, instead of a period of time representing the threshold condition for releasing the requests, the optimizing scheduler could be configured such that a specified number of accumulated requests serves as the threshold condition. For example, the optimizing scheduler could accumulate 100 requests before optimizing the requests. For this embodiment, optimizing scheduler 304 includes a counter which maintains a count of the number of read/write requests received from application 302.

According to one embodiment of the present invention, the optimizing scheduler 204 sorts the read/write requests so that the order of the requests corresponds to the order in which the sectors to be read are sequentially distributed on the disk. Thus, with reference to Figure 2, the optimum order of the sectors is 1-4-3-2, since these sectors are contained in sequential tracks from outermost to innermost track in this particular order. Furthermore, the sorting direction between sets of read/write requests is exchanged so that the drive mechanism sweeps across the disk in one direction first, then back in the other direction. This eliminates the need for the drive mechanism to move back to the opposite side of the disk after the innermost or outermost

track of the disk has been reached.

Figure 5 provides a table which summarizes the access request order and resulting number of seeks for the simple FIFO and two-command queue system described above, in comparison with the results obtained for a scheduler according to an embodiment of the present invention. The sector numbers provided in the Request Order column of Figure 5 correspond to the sector numbers illustrated in Figure 2.

With regard to the method in which the optimizing scheduler 304 sorts the order of the read/write requests, a simple sorting algorithm is implemented in which the sort operation is performed on all collected requests in one pass. Alternatively, the optimizing scheduler implements an incremental sorting algorithm in which requests are placed into their optimal order as they are received.

In a further alternative of the present invention, the read/write operations are prioritized, so that background operations can be performed without impacting the real-time performance of the system. As the distribution of the read/write operations approaches worst case, background read/write requests can be deferred to a later set of requests so that higher priority read/write operations are completed in time. For better case sets of read/write operations, background read/write requests could be interleaved (and still seek-optimized) without causing real-time problems.

Figure 4 is a flowchart illustrating the major steps of optimizing the execution order of disk read/write commands according to a method for one embodiment of the present invention, and with reference to the file system illustrated in Figure 3. In step 402, the optimizing scheduler 304 collects disk read/write requests sent from the application program 302 for one scheduler period. The scheduler period corresponds to the number of read/write requests which are held in the read/write queue before the order of the read/write requests is optimized. After the specified number of read/write requests are

collected in the optimizing scheduler buffer, the read/write requests are re-ordered so that the distance of seek operations between consecutive read/write operations is minimized, step 404. The read/write commands are then transmitted from the optimizing scheduler 304 to the drive controller 306 in the optimized order, step 406. If there are additional disk read/write requests issued from the application, the process repeats from step 402, otherwise the process ends.

As has been mentioned earlier, increasing the command queue size increases the latency between the time a read/write request is initiated and the time the operation is performed. This is not generally a problem for write operations and is not a problem for read operations if the disk locations to be accessed are known far enough in advance (as is usually the case for media tracks). Moreover, a larger queue requires larger data buffers than conventional queuing methods, however, this is often an acceptable trade-off for increased disk I/O bandwidth.

One embodiment of the present invention provides a file system method which is suitable for use with applications with high disk I/O bandwidth requirements in which excessive disk access time may negatively impact application performance. A method of the present invention provides a system for optimizing the order in which disk read/write commands are executed, so that disk seek operations may be reduced to satisfy the disk I/O bandwidth requirements of the application programs.

One exemplary application of the optimizing scheduler is its use in a computer implemented digital player/recorder including audio record and playback programs which read and write multiple tracks of data simultaneously to the disk. Such an application has rigorous minimum disk I/O bandwidth requirements for playback, and the disk head may be forced to seek among blocks on the disk in order to play a single track. In some instances, the data samples may not be read from the disk within the required time, in which case, the audio data played back may be interrupted or otherwise distorted. The ordering of large sets of read/write requests in accordance with the physical distribution

of sectors on the disk, as described in reference to Figure 4, minimizes the chance that applications fail due to excessive seek operations causing increased data transfer cycles.

Although the above discussion was written in the context of audio applications, it should be noted that a method of the present invention could be used in other similar applications, such as applications which involve both video and audio content, or applications which involve high-speed calculations on limited amounts of data.

In one embodiment of the present invention, the optimizing scheduler 304 illustrated in Figure 3 is implemented as a program which is executed by a processor coupled to the disk drive which is to be accessed (e.g., processor 102 in computer system 100). The optimizing scheduler program could be a program which is incorporated as part of the disk file system, or it could be a stand-alone program which can be called by the disk file system or an application program. Appendix A provides a detailed listing of C++ program code which implements an optimizing scheduler according to one embodiment of the present invention. It will be appreciated however, that methods of the present invention are not limited to the programming language and exact code sequences provided.

The steps of a method of the present invention may be implemented by a central processing unit (CPU) in a computer executing sequences of instructions stored in a memory. The memory may be a random access memory (RAM), read-only memory (ROM), a persistent store, such as a mass storage device, or any combination of these devices. Execution of the sequences of instructions causes the CPU to perform steps according to the present invention. The instructions may be loaded into the memory of the computer from a storage device or from one or more other computer systems over a network connection. Consequently, execution of the instructions may be performed directly by the CPU. In other cases, the instructions may not be directly executable by the CPU. Under these circumstances, the

instructions may be executed by causing the CPU to execute an interpreter that interprets the instructions, or by causing the CPU to execute instructions which convert the received instructions to instructions which can be directly executed by the CPU. In other embodiments, hardwired circuitry may be used in place of, or in combination with, software instructions to implement the present invention. Thus, the present invention is not limited to any specific combination of hardware circuitry and software, nor to any particular source for the instructions executed by the computer.

In the foregoing, a system has been described for collecting and optimizing the order of read/write requests in a disk file system. Although the present invention has been described with reference to specific exemplary embodiments, it will be evident that various modifications and changes may be made to these embodiments without departing from the broader spirit and scope of the invention as set forth in the claims. Accordingly, the specification and drawings are to be regarded in an illustrative rather than a restrictive sense.

Appendix A

```
// NAME qfssched.hpp
//
// DESCRIPTION
// Quick version of scheduler
//
// COPYRIGHT
// copyright 1994 by Advanced digital Systems Group.  All rights
reserved.
//
// VERSION CONTROL
//    $Header:    G:/PVCS/ARCHIVES/hchallel/QFSSCHED.HPV    1.16
      AUG 14, 1997 12:01:16   JCLAAR      $
//

#ifndef_QFSSCHED_HPP_
#define_QFSSCHED_HPP_

#include <vector>
#include <list>

#include <dbllnk.hpp>
#include "task.hpp"
#include <timedef.h>
#include <sndfile.hpp>
#include "refcount.hpp"

class SyIoObject;

class SyIoReferenceObject : public SyReferenceCount
{
     SyIoObject* mIoObject;
public:
     inline SyIoReferenceObject (SyIoObject* ioObject);
```

```
        -SyIoReferenceObject();
        inline void BuildIoList ( IoNodeList& ioList );
        inline SyIoobject* IoObject() const;
        inline SyQFS* FileSystem (void);
        inline void RemoveIoObject()
        void AddNotify();
);

class SyQuickScheduler : public SyTask
(
        SySync::SycriticalSection mObjAccess;
        SySync::SycriticalSection mReadyObjAccess;
        typedef STD::vector<SyIoObject*,
STD::allocator,SyIoObject*>> IoObjects;
        typedef STD::list<SyioReferenceObject*,
STD::allocator<SyIoReferenceObject*>> ReadyIoObjects;

        IoObjects mIoObject;
        ReadyIoObjects mReadyIoObject;
        UInt32 mPeriod;
        UInt32 mResolution;
        Bool  mSortForward; // Sort direction for I/Os

        void SortReadyList (IoNodeLists& listToSort);

protected:
        virtual UInt32 Iterate(Ubt16 index):
public:
        HCHAN_EXP SyQuickScheduler (Uint32 period, UInt32
resolution);
        HCHAN_EXP -SyQuickScheduler();
        virtual void AddObject (SyIoObject* object);
        virtual void RemoveObject (SyIoObject* object);
        Virtual void AddReadyObject (SyIoReferenceObject* object ):
        virtual void RemoveReadyObject (SyIoObject* object ):
        inline UInt32 Period() const;
```

```
        inline UInt32 Resolution() const;
        inline TimeCode IoPeriod() const;
};
#inline "qfssched. inl"

#endif


// NAME qfssched.cpp
//
// DESCRIPTION
//    Quick version of scheduler
//
// COPYRIGHT
//    copyright 1997 by Advanced Digital Systems Group.  All right
reserved.
//
// VERSION CONTROL
//    $Header:    G:PVCS/ARCHIVES/hchallel/qfssched.cpv     1.37
      AUG 22, 1997 14:03:24   jclaar      $
//

#include "precomp.h"
#include "qfssched.hpp"
#include <mutex.hpp>
#include <algorithm>

void SyIoReferenceObject::AddNotify()
{
      mIoObject->AddNotify();
}

SyIoReferenceObject::~SyIoReferenceObject()
{
      _ASSERT(this->GetRefernceCount() == 0);
      //TRACE(_T("SyIoRefobj-Datrct: %ld\"), this );
```

```
)
SyQuickScheduler::SyQuickScheduler
(
      UInt32 period.
      Uint32 resolution
)
      : SyTask
            (
            NULL
            NULL
            0,
            FALSE,
            period.
            true,
            kEBxSchedulerPriority
      ),
      mPeriod(period),
      mResolution(resolution).
      mSoftForward(TRUE)
(
)

SyQuickScheduler::-SyQuickScheduler()
(
      if(!mReadyToObject.empty() )
(
//    TRACE(_T("qsch-dstrct size; %ld\"), mReadyIoObject->size()
);
      for( ReadyIoObjects::iterator iter = mReadyIoObject.begin();
      iter 1= mReadyIoObject.end();
      iter++
          )
      (
//    TRACE(_T("qsch-dstrct: %ld\n", (*iter) );
      (*iter)->Decrement():
           )
```

```
        )
        this->kill():
)

void SyQuickScheduler::AddObject(SyIoObject* object)
(
        _ASSERT (object 1= NULL;
        this->Stop():
        (
        GrabCriticalSection cs (&mObjAccess);
        // don't add the object if it is already in here.
        _ASSERT(STD::find(mIoObject.begin(). mIoObject.end().
object) ==mIoObject.end());
        mIoObject.push_back (object);
        this->SyTask::AddObject(object-.Event());
        )

void SyquickScheduler::RemoveObject (SyIoObject* object)
(
        _ASSERT(object 1= NULL);
        this-.Stop():
        (
        GrabCriticalSection cs(&mObjAccess);
        IoObjects::iterator i =
        STD::find(mIoObject.begin()), mIobject.end(), object):
        _ASSERT(1 1 mIoObject.end());
        mIoObject.erase (i);
        this-.SyTask::RemoveObject->Event()):
        )
        this->Start():
)

void SyQuickScheduler::AddReadyObject (SyIoReferenceObject*
object)
(
        //
```

```
        // this code protects against duplicate ioobjects. but
        // with the new scheme, I don't think duplicate ioobjects
        //can ever happen.  This could save some time (RMD)
        //
        GrabCriticalSection cs (&mReadyObjAccess);
        _ASSERT(object != NULL);
        // Don't add the object of it is already in here.
        if)object->GerReferenceCount() > 2) );
        //TRACE(_T("AddReady - cnt %ld\n"), object-
>GetReferenceCount() ),
        return:
        )
        //if (STD::find(mReadyIoObject->begin(). mReadyIoObject-
.end(). object)
        //return;
        mReadyIoObject.push_back(object);
        object->Increment();
        object-> AddNotify();
        //TRACE)_T("QS-ARO:%ld\n"), object );
void SyQuickScheduler::RemoveReadyObject(SyIoObject" object)
(
        If ((*Iter)->IoObject() == object)
        )
        else

           iter++;
        )
)
UInt32 SyquickScheduler::Iterate(Knt16 index)
(
        UInt32 start = SyTimeGetTime():

        //
        //get the file system that io's will be done on by
        //quering the io object that was signaled
        //
```

```
        _ASSERT(mIoObject. size() == this->NumObjects()):
#ifdef _DEBUG
        if (index != WAIT_TIMEOUT)
        (
        _ASSERT(index.=0 index, mIoObject.size()):
        _ASSERT(this->WaitObjects()[index] == mIoObject [index]-
>Event()0;
        )
#endif

        // Don't need to get the mObjAccess critical section since
AddObject
        // and RemoveObject stop the scheduler.
        SyQFS* pQFS = NULL;
        if(index != WAIT_TIMEOUT)
        (
        pQFS =mIoObject[index]->FileSystem();
        )
        else if( !mReadyIoObject.empty())
        (
        mReadyObjAccess.Enter():
        pQFS = mReadyIoObject.front()->FileSystem():
        mReadyObjAccess.Leave():
)
        //TRACE(_T("qsch-index: %ld,readysize: %ld. pQFS: %ld,
priority: %ld\n"), index, mReadyIoObject->size(),
        pQFS, GetThreadPriority(this->Thread() ) );
        //if(index < mIoObject->size() )
        //TRACE(_T("qsch-iter eventsig: %ld\n"), (mIoObject)
[index]->Event() );

        _ASSERT() (pQFS ==NULL && index != WAIT_TIMEOUT));
        if( pQFS 1= NULL)
        (
        //
```

```
        //copy the current list to a temp list for those items that
        //match the designated filesystem.  Protect access to list
        //
        //NOTE: The following code assumes that there is one
scheduler
        // per disl volume.  If there is not, the code below this
will
        // assert.
        ReadyIoObjects tempList;
        mReadyObjAccess.Enter();
        // splice removes all objects from mReadyIoObject.
        tempList. splice(tempList.begin(), mReadyIoObject);
        mReadyObjAccessLeave();

#ifdef_DEBUG
        //verify that all objects are really on the same file
system.
        (
        ReadyToObjects:: iterator iter = tempList.Begin();
        for ( ; iter != tempList.end(); iter++)
            _ASSERT((*iter)->Filesystem() == pQFS);
        )
#endif

        //
        //iterate over the temp list and call BuildIoList for all
objects in
        //the temp list.  if the temp list is empty, then reset the
event that
        //triggered the scheduler(prevents endless reentry)
        //
        ToNodeList readyIoList;

        if(tempList.empty() )
        (
        if(index != WAIT_TIMEOUT)
```

```
        (
        ::SyResetEvent(mIoObject(index)->Event() );
        )
        )
        else
        (
        ReadyIoObjects::iterator iter;
        for(iter = TempList.begin(): iter != tempList.end(); iter++)
        (
        //TRACE(_T("QS~BIOL %ld\n"), (*iter) _:
        (*iter)->BuildIoList(readyIoList):
        )
        )
        // Sort the list by volume offset
        if (!readyIoList.empty())
        (
        this->SortReadyList(readyIoList);
        IoNodeList::iterator i = readyIoList.begin();
        for (; 1 != readyIoList.end(); i++)
           pQFS->PerformAsynchronousIO((*i).get());

        )
)
UInt32 last = SyTimeGetTime() - start;
this-.SetTimeout(( this-> period() > last) ? this->period() - last
: 0, FALSE );

return D;
)
void SyQuickScheduler::SortReadyList (IoNodeList& listToSort)

      //Sort the lost bsed on the current value of mSortForward
      if (mSortForward)
          listToSort.sort (STD::greater<SyAutoIoNode());
      else
          listToSort.sort();
```

```
        // Next time, sort the other way.
        mSortForward = !mSortForward;



/ NAME qfssched.inl
//
// DESCRIPTION
//    Quick version of scheduler
//
// COPYRIGHT
//    copyright 1994 by Advanced Digital Systems Group.  All right
reserved.
//
// VERSION CONTROL
//    $Header:    G:PVCS/ARCHIVES/hchallel/qfssched.inv     1.4
      Dec 23, 1996 14:21:38   RMD   $
//

#ifndef_QFSSCHED-INL-
#define_QFSSCHED-INL_

#include "ioobject.hpp"

//SyQuickScheduler
//
inline UInt32 SyQuickScheduler::Period() const
(
      return mPeriod;
)
inline UInt32 syQuickScheduler::Resolution() const
(
)
inline Timecode syQuickScheduler::IoPeriod() const
(
```

```
        return 0x8000;
    )

    //
    //SyIoReferenceObject
    //
    inline SyIoReferenceObject::SyIoReferenceObject
    (
        SyIobject* ioObject
    )
        ; SyReferenceCount(),
        mIoObject (ioObject)
    (
    )

    inline void SyIoReferenceObject::BuildIoList
    (
        IoNodeList& ioList
    )
    (
        //
        // assign member variable locally because Decrement() has
        // the potential to delete 'this'. Even through 'this' is
    passed
        // into BuildIoList(), it is only used for comparative
    purposes
        /
        SyIoObject*       pIoObject         = mIoObject;
        this->Decrement();
        pIoObject->BuildIoList (ioList, this );
    inline SyIoObject* SyIoReferenceObject::IoObject() const
    (
        return mIoObject;
    )
    inline SyQFS*SyIoReferenceObject::FileSystem (void)
    (
```

```
        return (mIoObject ! =NULL /*&& this->GetReferenceCount() >
1*/ ) ? mIoObject->FileSystem() : NULL;
)

inline void SyIoReferenceObject::RemoveIoObject()
(
        mIoObject = NULL;
)
#endif
```

CLAIMS

What is claimed is:

1. A method of executing disk access requests issued by an application program, the method comprising the steps of:

receiving a plurality of disk read/write requests from said application program;

collecting a set of said plurality of read/write requests until a predetermined threshold condition is reached;

sorting the order of said set of read/write requests in relation to the physical distribution on a disk of sectors to be accessed by each read/write request of said set of read/write requests; and

transmitting said read/write requests to a drive controller controlling said disk in the order created in said sorting step.

2. A method according to claim 1 wherein said set of read/write requests are stored in a buffer memory coupled to said drive controller.

3. A method according to claim 2 wherein said collecting step further comprises blocking said plurality of read/write requests from being transmitted from said application to said drive controller for a predetermined period of time.

4. A method according to claim 2 wherein said collecting step further comprises blocking said plurality of read/write requests from being transmitted from said application to said drive controller until a predetermined number of read/write requests is accumulated in said buffer.

5. A method according to claim 3 wherein said sorting step is performed on said set of read/write requests upon the collection of an entire set of read/write requests within said time period.

6. A method according to claim 3 wherein said sorting step is

performed on each read/write request as it is received into said buffer.

7. An apparatus for optimizing the order of disk access commands issued by an application program, said apparatus comprising:

a buffer receiving a plurality of read/write requests issued by said application program, said buffer receiving said plurality of read/write requests until a threshold condition is reached, said read/write requests received when said threshold condition is reached defining a set of read/write requests;

a scheduler logically coupled to said buffer, said scheduler configured to sort said set of read/write requests wherein the order of said set of read/write requests corresponds to the physical distribution on a disk of sectors to be accessed by each read/write request of said set of read/write requests; and

a drive controller logically coupled to said scheduler for controlling access to said disk in accordance with said set of read/write requests.

8. An apparatus according to claim 7 wherein said scheduler blocks said plurality of read/write requests from being transmitted from said application to said drive controller for a predetermined period of time.

9. An apparatus according to claim 7 wherein said collecting step further comprises a counter, said counter maintaining a count of read/write requests received by said buffer, said scheduler configured to block said plurality of read/write requests from being transmitted from said application to said drive controller until said counter reaches a predetermined number of requests.

10. An apparatus according to claim 8 wherein said scheduler sorts said set of read/write requests upon the collection of an entire set of read/write requests within said period of time.

11. An apparatus according to claim 8 wherein said sorting

step is performed on each read/write request as it is received into said buffer.

12. An apparatus for executing disk access requests issued by an application program, the method comprising the steps of:

means for receiving a plurality of disk read/write requests from said application program;

means for collecting a set of said plurality of read/write requests until a predetermined threshold condition is reached;

means for sorting the order of said set of read/write requests in relation to the physical distribution on a disk of sectors to be accessed by each read/write request of said set of read/write requests; and

means for transmitting said read/write requests to a drive controller controlling said disk in the order created in said sorting step.

13. An apparatus according to claim 12 further comprising buffer means for storing said set of read/write requests.

14. A computer readable medium having stored thereon sequences of instructions which are executable by a processor, and which, which, when executed by the processor, cause the processor to perform the steps of:

receiving a plurality of disk read/write requests from an application program;

collecting a set of said plurality of read/write requests until a predetermined threshold condition is reached;

storing said set of read/write requests in a buffer memory;

sorting the order of said set of read/write requests in relation to the physical distribution on a disk of sectors to be accessed by each read/write request of said set of read/write requests; and

transmitting said read/write requests to a drive controller controlling said disk in the order created in said sorting step.

15. A computer according to claim 14 wherein the memory further contains instructions which cause the processor to perform the step of blocking said plurality of read/write requests from being

transmitted from said application to said drive controller for a predetermined period of time.

16. A computer according to claim 14 wherein the memory further contains instructions which cause the processor to perform the step of blocking said plurality of read/write requests from being transmitted from said application to said drive controller until a predetermined number of read/write requests is accumulated in said buffer.

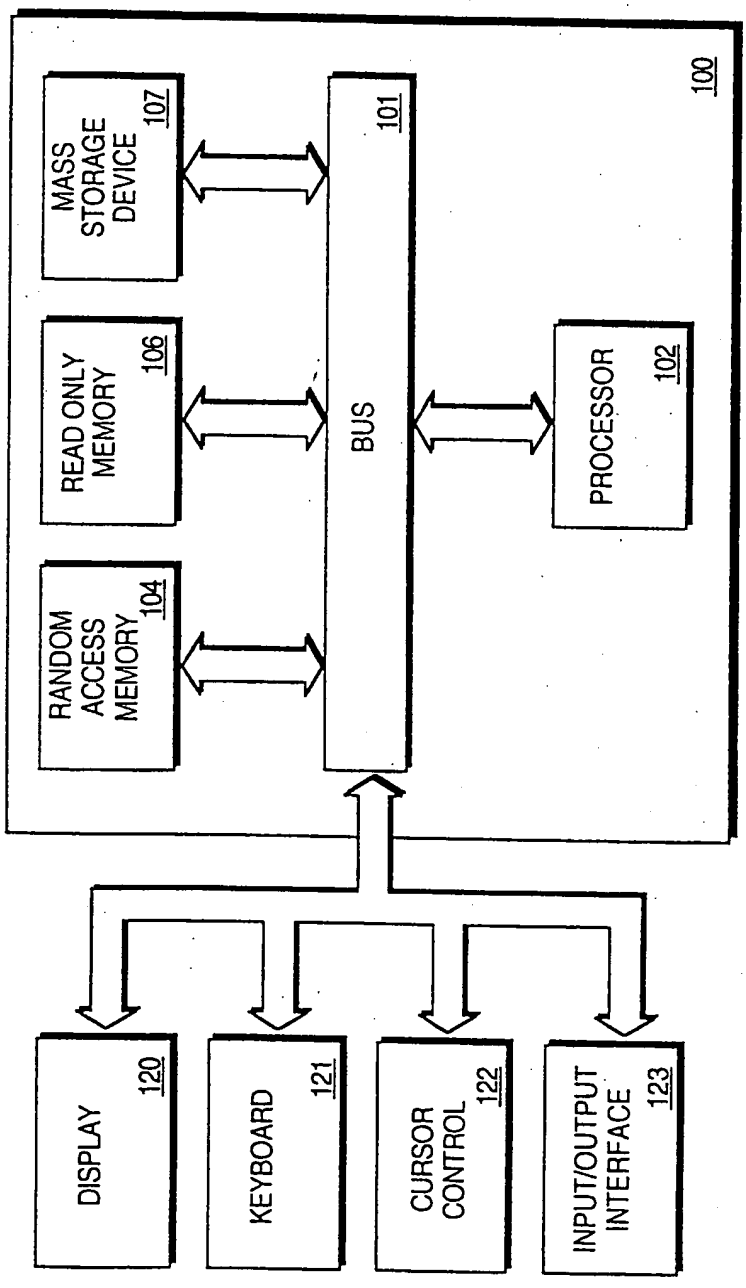

100
MASS STORAGE DEVICE 107
READ ONLY MEMORY 106
RANDOM ACCESS MEMORY 104
BUS 101
PROCESSOR 102
DISPLAY 120
KEYBOARD 121
CURSOR CONTROL 122
INPUT/OUTPUT INTERFACE 123

FIG. 1

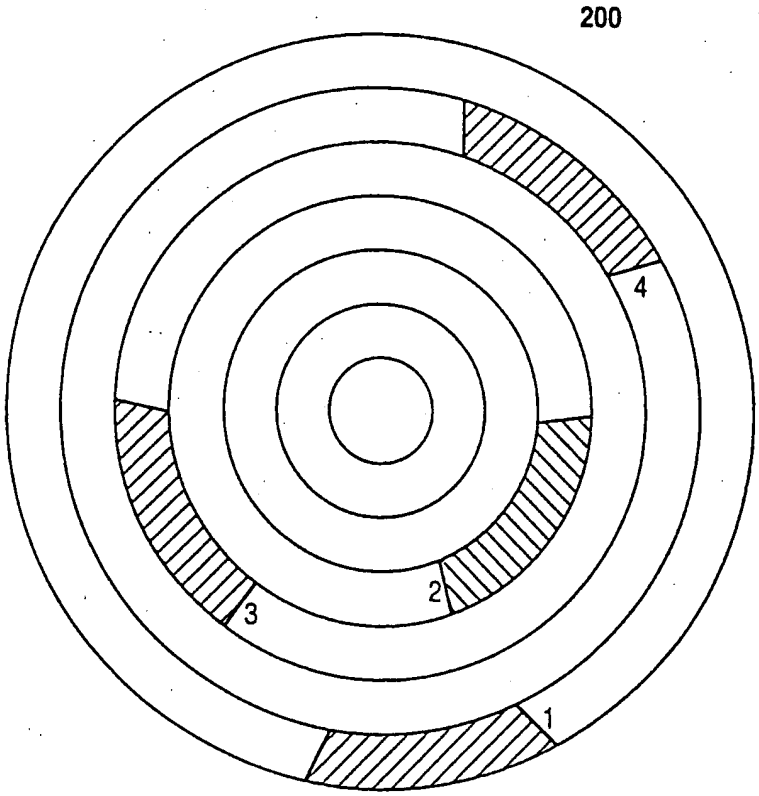


FIG. 2

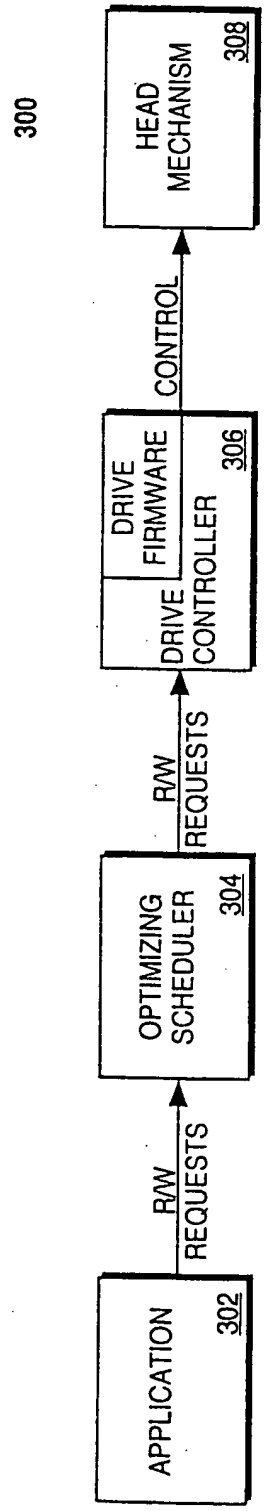

300
APPLICATION
302
R/W REQUESTS
OPTIMIZING SCHEDULER
304
R/W REQUESTS
DRIVE CONTROLLER
DRIVE FIRMWARE
306
CONTROL
HEAD MECHANISM
308

FIG. 3

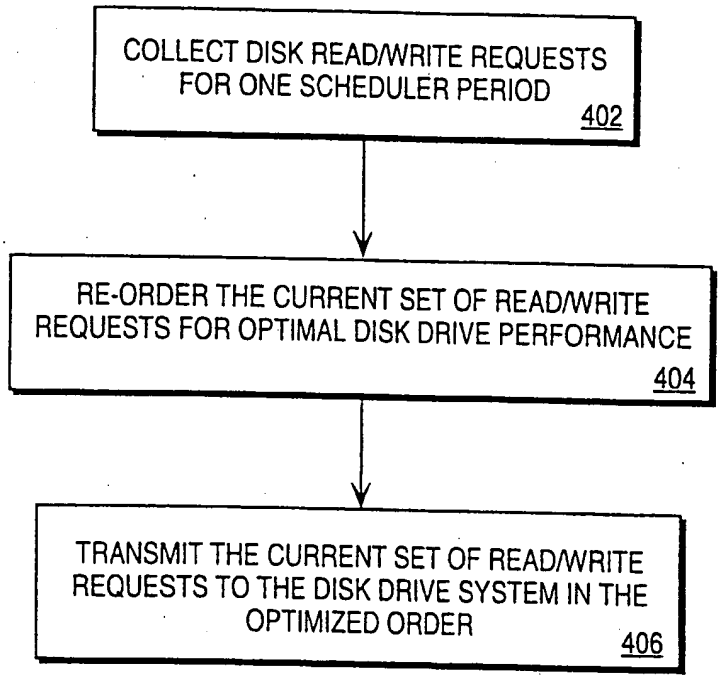

COLLECT DISK READ/WRITE REQUESTS FOR ONE SCHEDULER PERIOD
402
RE-ORDER THE CURRENT SET OF READ/WRITE REQUESTS FOR OPTIMAL DISK DRIVE PERFORMANCE
404
TRANSMIT THE CURRENT SET OF READ/WRITE REQUESTS TO THE DISK DRIVE SYSTEM IN THE OPTIMIZED ORDER
406

FIG. 4

ORDERING METHOD	REQUEST ORDER	TOTAL SEEK DISTANCE
Simple FIFO	1-2-3-4	5 Tracks
Command Queuing (two requests queued while one completes)	1-3-2-4	5 Tracks
Scheduler (all I/O's within one period queued)	1-4-3-2 (Optimal)	3 Tracks

FIG. 5

INTERNATIONAL SEARCH REPORT

Interr. ...al Application No
PCT/...8/18441

A. CLASSIFICATION OF SUBJEC...TER
IPC 6 G06F3/06

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6 G06F G11B

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
X	WO 97 16783 A (SONY CORP) 9 May 1997 see page 28, line 12 - page 29, line 9 see page 38, line 12 - page 40, line 2; figures 2,4	1,2,4,7, 9,12-14, 16

X	PATENT ABSTRACTS OF JAPAN vol. 004, no. 142 (P-030), 7 October 1980 & JP 55 091049 A (FUJITSU LTD), 10 July 1980	1-3,7,8, 12-15
A	see abstract	4-6, 9-11,16

	-/--	

[X] Further documents are listed in the continuation of box C.

[X] Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
15 December 1998	21/12/1998

Name and mailing address of the ISA	Authorized officer
European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040, Tx. 31 651 epo nl, Fax: (+31-70) 340-3016	Moens, R

Form PCT/ISA/210 (second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Inter nal Application No
PCT 98/18441

C.(Continuation) DOCUMENTS C ERED TO BE RELEVANT

Category *	Citation of document, with indication,where appropriate, of the relevant passages	Relevant to claim No.
X	US 5 644 786 A (GALLAGHER MICHAEL J ET AL) 1 July 1997 see column 2, line 8 - column 4, line 51; figures 3,5	1,2,4,7, 9,12-14, 16
A	"QUEUING ACCESS REQUESTS TO DIRECT ACCESS STORAGE DEVICE" IBM TECHNICAL DISCLOSURE BULLETIN, vol. 38, no. 7, 1 July 1995, pages 423-425, XP000521743 see the whole document	1-16

Form PCT/ISA/210 (continuation of second sheet) (July 1992)

INTERNATIONAL SEARCH REPORT

Inter[redacted]nal Application No
[redacted]/US 98/18441

Patent document cited in search report	Publication date	Patent family member(s)	Publication date
WO 9716783 A	09-05-1997	AU 7336396 A	22-05-1997
		BR 9607550 A	07-07-1998
		CA 2207076 A	09-05-1997
		CN 1166883 A	03-12-1997
		EP 0806003 A	12-11-1997
		JP 9185864 A	15-07-1997
		PL 322016 A	05-01-1998
		US 5708632 A	13-01-1998
US 5644786 A	01-07-1997	NONE	